Student ID : 10533133
Name : 唐盛銘
Checkpoint : ppc5

Words in bold mean that they are variable names or key points.
[ ] : topic of the following paragraphs.

[0] System setup :

Keep 2 variables for time record purpose, one called **time_quantum(initialized to -1)**, got incremented every time timer interrupt comes, and modulo 4 after incremented so **time_quantum** will only have values 0, 1, 2, and 3, Another one called **time_elapsed**, got incremented every time it sees that **time_quantum == 0** since **I choose 4 times of Edsim51 timer overflow as one time unit for delay**.

I have a semaphore called **time_sem** and an array of 3 characters called **time_remain[3]** to store the remaining time for each thread they need to wait until return. **time_sem** is used to protect the update of **time_remain[3]** since each thread get its delay call independently. Note that thread k stamps their remaining time at **time_remain[k - 1]**. Another character **time_delay_record** is used to stamp each thread's time_quatum on. The bit-level layout in this variable, for example, would be

time_delay_record : 01_10_11_00, means that
thread 3 called delay() at time_quantum == 1
thread 2 called delay() at time_quantum == 2
thread 1 called delay() at time_quantum == 3
thread 0 called delay() at time_quantum == 0
However, thread 0 is preserved for thread manager (can achieve this by create this thread first), so this thread is not user-reachable (User's thread ID starts from 1 to 3).

Use of **thread_bm**:
The lower four bits of **thread_bm** are used to mark that the thread is being used or not. The higher four bits are used to mark that if the thread is waiting for not.
For example, if
thead_bm = 1000_1011, means that
lower part : xxxx_1011
→ thread 3, 1 and 0 are used.
higher part : 1000_xxxx
→ thread 3 is actually waiting, and has higher priority to be chosen to run than any other thread that is not waiting.

The system has a routine that every time the PC jumps to myTimer0Handler(), it will poll every 2-bit group in **time_delay_record** to see if the current **time_quantum** matches the corresponding 2 bits.

Use the previous example, suppose that if the current **time_quatum** is 1 then **time_remain[2]** will decrease by one.

time_delay_record = 01_xx_xx_xx matches 1 (01 part).

time_delay[3] = {0x08, 0x05, 0x01} → {0x08, 0x05, 0x00}, means that

thread 1 still has to wait for 8 time unit

thread 2 still has to wait for 5 time unit

thread 3 has finished waiting and is the next thread picked to run if it

Little modification on ThreadYield():

In the last project checkpoints, we use round-robin policy, however, simply round-robin will not preserve the precision of delay() since threads may run at other **time_quantum** instead of the one it stamps on **time_delay_record** if the number of used thread is not 4, and threads only have the chance to return from delay() at **its time quantum**. So, after round-robin, looking for threads that finished waiting this round, and overwrite the result in round-robin if such one exists. If there is no thread finishes waiting, the whole runs as usual like the previous checkpoints.

[1-1] Implementation of delay(n) function:

1. Wait for **time_sem**
2. Stamp the current **time_quantum** onto **time_delay_record** according to thread ID after clearing the corresponding 2 bits in **time_delay_record**.
3. Set the corresponding high part of **thread_bm** to mark that the thread has entered delay().
4. Set **time_remain[thread_id - 1]** to the time it needs to delay.
5. Signal **time_sem**
6. Poll to check if **time_remain[thread_id - 1]** is 0. (0 means it finishes waiting)
7. Clear the record in **time_delay_record** and in **thread_bm**(mark that it is not waiting).
8. Return.

[1-2] Implementation of now() function:

1. If the current time_quantum < 2, return **time_elapsed,** otherwise return **time_elapsed** + 1 (round to next time unit)

Note that I don't need to use this function to implement delay(), but I implemented it anyway.

[2] Robust thread termination and creation:

Semaphore **thread_ct** is initialized to 4 (max thread supported)

For ThreadCreate():

1. Wait for semaphore **thread_ct**.
2. Pick a thread ID for this created thread.
3. Initialize stack pointer to corresponding address.
4. Push the address of ThreadExit() before pushing any other registers.
5. Push A, B, … like the previous checkpoints.

Pushing the address of ThreadExit() so that we don't need to manually call it every time a function finishes.

For ThreadExit():

1. Clear the corresponding bit in **thread bitmap**
2. Signal semaphore **thread_ct**
3. Enter an loop and wait for context switch

These steps are safe though the terminated thread stuck at a loop since it won't be picked to run in the next round. It is safe to do so but is not necessary.

Assemly code to push address of threadExit(), fp and other registers. The PUSH _i is to push PSW (i has been preprocessed so that it can put into stack directly)

```
__asm
    MOV A, DPL // store function ptr
    MOV B, DPH // store function ptr
    MOV DPTR, #_ThreadExit
    PUSH DPL // address of ThreadExot so that it can safety quit
    PUSH DPH
    MOV DPL, A // restore function ptr
    MOV DPH, B // restore function ptr
    PUSH DPL // address of fp so that it can resume
    PUSH DPH
    MOV A, #0
    PUSH ACC // ACC
    PUSH ACC // B
    PUSH ACC // DPL
    PUSH ACC // DPH
    PUSH _i
    MOV A, #0x00 // reset A, dummy
__endasm;
```

[3] Parking lot example:

Keep a semaphore called **slot (initialized to 2)** and a character called **spot** to represent the usage of the 2 lots.

The bit-layout of **spot** will be, for example

spot = 0010_0100

The lower part : xxxx_0100 means that the first lot is occupied by the fourth car.

The higher part : 0010_xxxx means that the second lot is occupied by the second car.

0000 means the lot is currently empty, since thread 0 is reserved for thread manage, any user thread won't get this thread ID.

For main(), simply initialized variable and semaphore, create threads for each car then enter an infinite loop (entering loop is not necessary).

Write 5 functions called CarA, CarB, CarC, CarD, CarE respectively.

For each car, do

1. Wait for semaphre **slot**
2. If the first lot is empty(use bit-wise AND), go to step 3, otherwise go to step 4
3. Write capital letter to buffer, call delay(), Write capital letter to log before return
4. Write small case letter to buffer, call delay(), Write small case letter to log before return

Note that the read/write operation need to wait and signal semaphore **mutex(initialized to 1)** and **empty(initialized to the size of buffer)**

.

There won't be two threads that finishes waiting at the same **time_quantum** (not time unit). So the problem of two cars coming out of lots in the same time does not exist , here the "same time" means time quantum. In terms of time unit, they may yield the lots in the same ***time unit*** but the ***relative order is clear*** since their time stamps on ***time_delay_record*** *are different (Every time interrupt comes, only one of the three characters in* ***time_remain[ ]*** *has a chance to be decremented).*

[extra] Use UART to write log:

Just like the Producer-Consumer example, in this case, producers are the 5 cars and myTimer0Handler(). myTimer0Handler() will write an 'X' to buffer every time unit (4 time quantum) at **time_quantum** == 0. myTimer0Handler() will consume one character every two time quantum (If you consume one character every time quantum, you will sometimes see some garbage out but the memory dump is correct).

Buffer size has minimum size about 6 or 7 since the read/write runs at different speed. You may see that a car wants to delay for one time unit but you see two 'X's between the marks if you actually run my code. That is normal because you should interprete 'X' as "time point" instead time unit (period). The time unit is the space between two 'X's. Another reason is that 'X' gets write to log first and the decrease the corresponding **time_remain[ ]** so if a car finishes waiting at time quantum 0 **,** you will see an 'X' first then it leaves. <u>You won't see a car just delays for k time units and more than k + 1 'X's between its enter and leave log.</u>

For example:

XAbXAbXXXX….. means that A, B both get their lots in the first time unit and both leave in the second time unit. The following time units contain no events.

[4] Typescript and Screen Shot:

Change the content in Makefile as the picture shows:

```
#
# makefile for testing cooperative multithreading
#
# This assumes you have SDCC installed and this targets EdSim51.
# The generated .hex file should be one that can be loaded and run
# directly.
#
# Author: Pai Chou
# CS 3423 Fall 2019
#

CC = sdcc
CFLAGS = -c
LDFLAGS =
#--stack-after-data --stack-loc 0x39 --data-loc 0x20

C_OBJECTS = testparking.rel preemptive.rel

all: testparking.hex

testparking.hex:   $(C_OBJECTS) $(ASM_OBJECTS)
				$(CC) $(LDFLAGS) -o testparking.hex $(C_OBJECTS)

clean:
	rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym

%.rel:  %.c    preemptive.h Makefile
	$(CC) $(CFLAGS) $<
```

$ cd target_direcoty

$ make clean

$ make

Open Edsim51 and load the “testparking.hex” file. Click Run !

Memory layout of preemptive.c (0x20~0x32) :

```
// fair version
#include <8051.h>
#include "preemptive.h"
static __data __at(0x20) char SP_saved[4];
static __data __at(0x24) char thread_id;
static __data __at(0x25) char thread_bm;
static __data __at(0x26) char thread_ct;
__data __at(0x27) char i;
static __data __at(0x28) char tmp;
static __data __at(0x2a) char last_thread;
// for delay
static __data __at(0x2b) unsigned char time_elapsed;
static __data __at(0x2c) char time_quatum;
static __data __at(0x2d) char time_delay_record; // aa-bb-cc-dd, each group denotes the time_quatum of each thread
static __data __at(0x2f) char time_sem;
static __data __at(0x30) char time_remain[3];
```

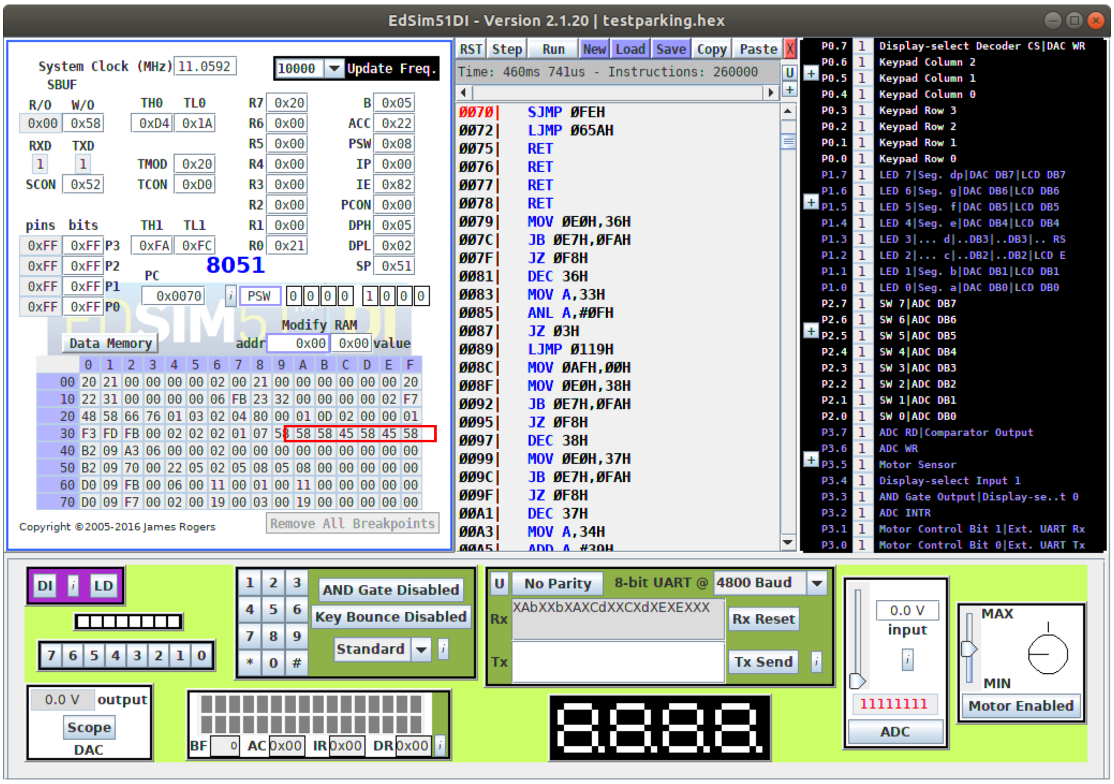
Memory layout of testparking.c (0x33~0x3f) :

```
#include <8051.h>
#include "preemptive.h"
#define A_DELAY 3
#define B_DELAY 2
#define C_DELAY 2
#define D_DELAY 3
#define E_DELAY 1

extern char i;
__data __at(0x33) char spot;
__data __at(0x34) char in;
__data __at(0x35) char out;
// semaphores
__data __at(0x36) char slot;
__data __at(0x37) char mutex;
__data __at(0x38) char empty;

// bounded buffer
__data __at(0x39) char buff[BUFF_SIZE];
```

The moment E has left for a while.



Example : The order of thread creation is A, B, C, D, E.

CarA delays 3 time units.

CarB delays 2 time units.

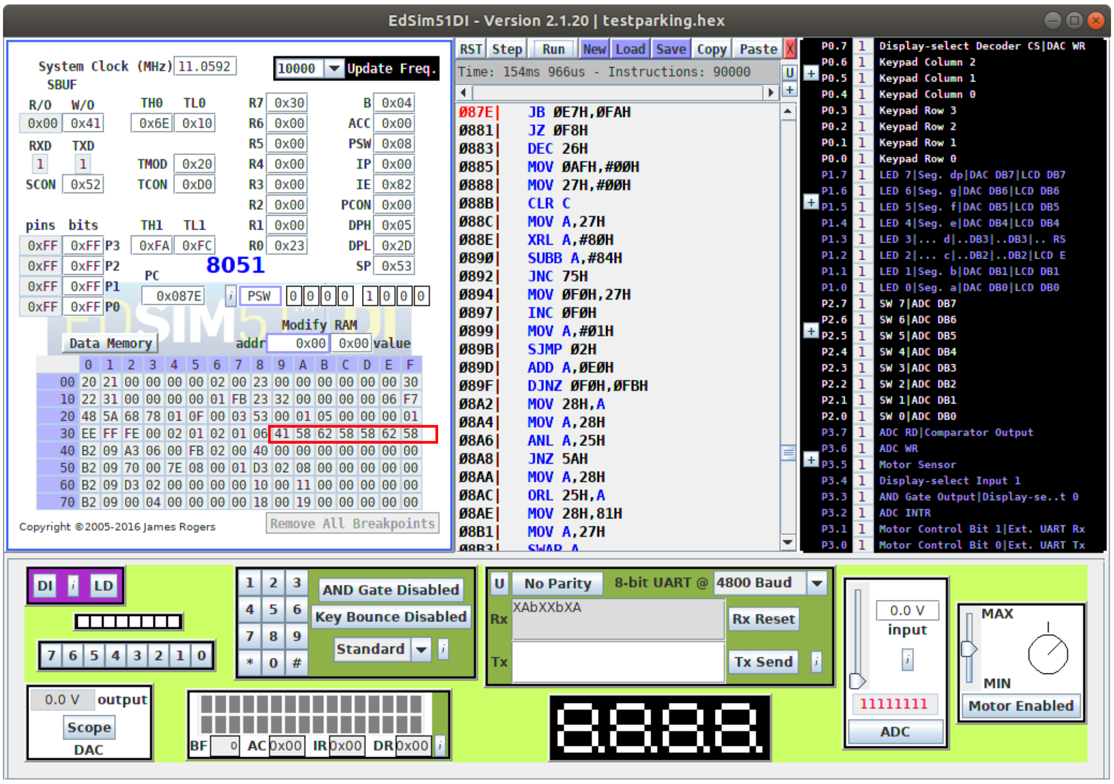
CarC delays 2 time units.

CarD delays 3 time units.

CarE delays 1 time units.

0x39 ~ 0x3f is the buffer space, the output from UART is relatively same in buffer. Relatvie order is correct but asynchronous output. See the two 0x45('E'), there are still some old records left, and most of them are placed with newly created 0x58('X').

The moment B has left for a while and A just left.



The one 0x41('A') means the left mark of A has just been written to this buffer, and the two 0x62('b') are the old records Car B has left in buffer.